embOS

Real-Time Operating System

CPU & Compiler specifics for embOS-Base Simulation using Visual Studio

Document: UM01060
Software Version: 5.18.3.0
Revision: 0
Date: October 28, 2024

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	support@segger.com*
Internet:	www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: October 28, 2024

Software	Revision	Date	By	Description
5.18.3.0	0	241028	MC	New software version.
5.18.0.0	0	220929	MM	Added chapter “Libraries”. Updated chapter “Build your own application”.
5.16.0.0	0	220105	TS	New software version.
5.14.0.0	0	210611	TS	New software version.
5.10.1.0	0	200703	MM	New software version.
5.8.2.0	1	200115	MM	Renamed and improved chapter “Calling blocking non-embOS functions from tasks”.
5.8.2.0	0	200113	MM	New software version.
5.06.1	0	190926	MM	New API function `OS_SIM_DeleteISRThread()`.
5.06	0	190812	MM	New software version. Added chapter “Updating the Simulation”. Added chapter “Handling Windows Callbacks”.
5.02	0	180724	MM	New software version. Updated chapter “Calling Windows API functions from tasks”.
4.40	0	180122	MC	New software version.
4.36	0	170809	TS	New software version.
4.34	0	170405	TS	New software version.
4.24	0	160629	TS	New software version.
4.22	0	160503	RH	New software version.
4.12a	0	150917	TS	New software version.
4.06a	0	150316	SC	First version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Using embOS Simulation

1.1 Installation

embOS is shipped as a zip-file in electronic form.

To install it, proceed as follows:

Extract the zip-file to any folder of your choice, preserving the directory structure of this file. Keep all files in their respective sub directories. Make sure the files are not read only after copying.

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find a prepared sample start project, which you should use and modify to write your application. So follow the instructions of section *First Steps* on page 11.

You should do this even if you do not intend to use the IDE for your application development to become familiar with embOS.

If you do not or do not want to work with the IDE, you should: Copy the library-file to your work-directory. The advantage is that when switching to an updated version of embOS later in a project, you do not affect older projects that use embOS, too. embOS does in no way rely on an IDE, it may be used without the IDE using batch files or a make utility without any problem.

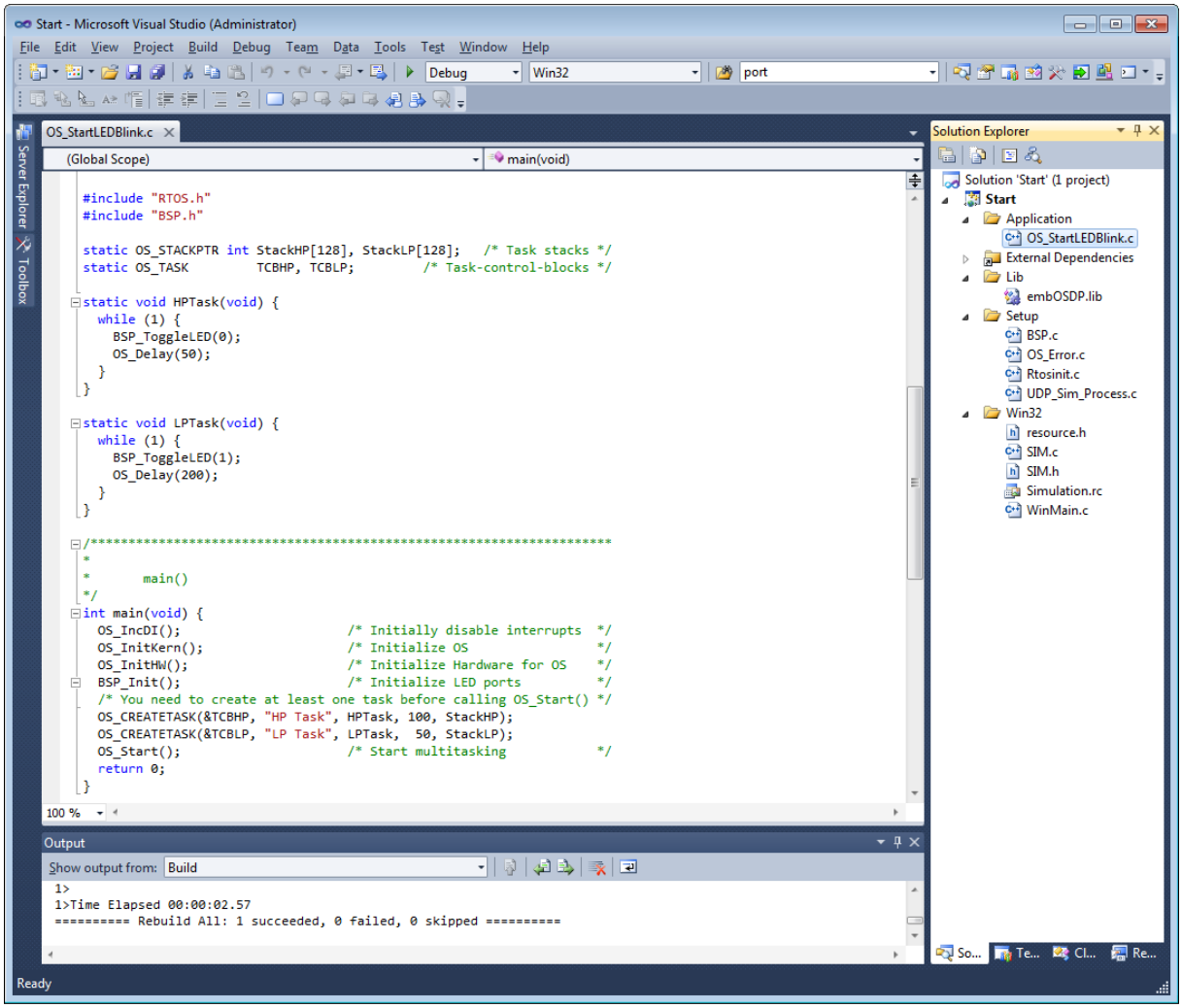
1.2 First Steps

After installation of embOS you can create your first multitasking application. You have received one ready to go sample start project and every other files needed in the subfolder `Start`. It is a good idea to use it as a starting point for all of your applications. The sample project is contained in the subfolder `BoardSupport`.

To get your new application running, you should proceed as follows:

- Create a work directory for your application, for example `c:\work`.
- Copy the whole folder `Start` which is part of your embOS distribution into your work directory.
- Clear the read-only attribute of all files in the new `Start` folder.
- Open the sample project in `Start\BoardSupport\Simulation` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

After generating the project of your choice, the screen should look like this:



1.3 The example application OS_StartLEDBlink.c

The following is a printout of the example application `OS_StartLEDBlink.c`. It is a good starting point for your application. (Note that the file actually shipped with your port of embOS may look slightly different from this one.)

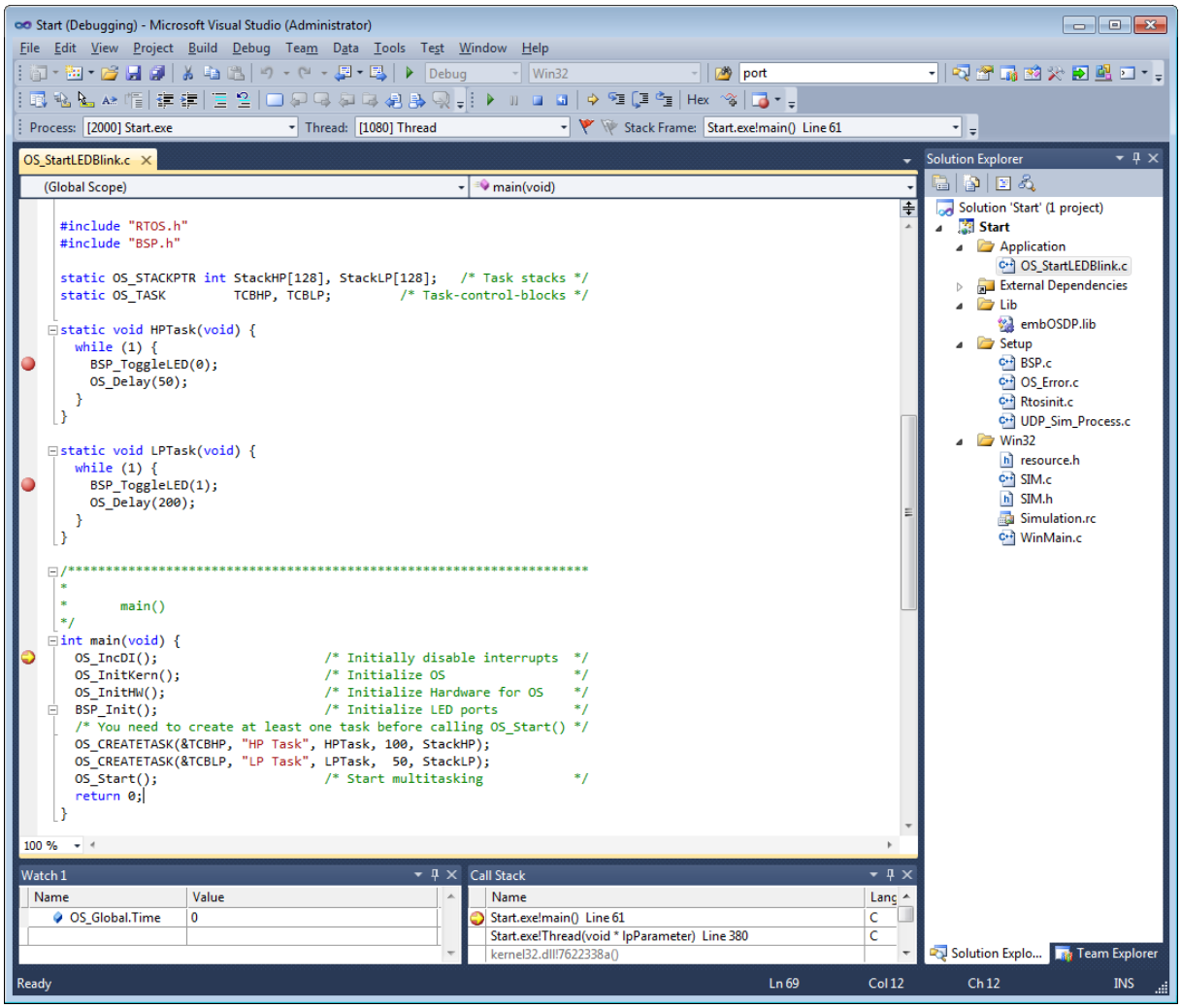
What happens is easy to see:

After initialization of embOS two tasks are created and started. The two tasks are activated and executed until they run into the delay, suspend for the specified time and continue execution.

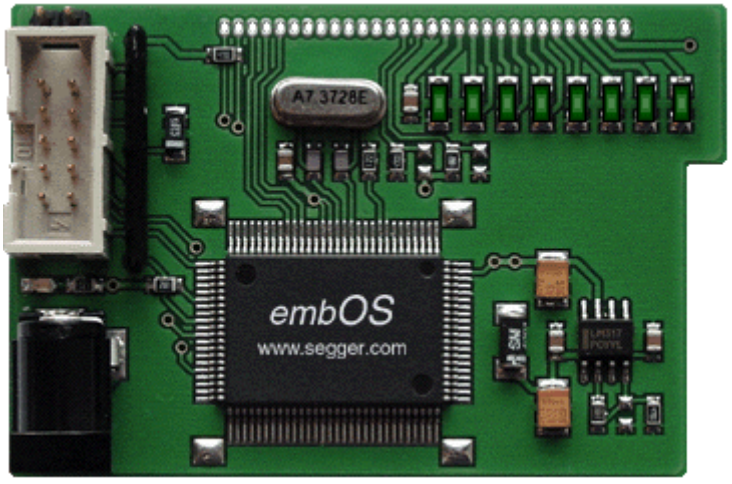
```
/*********************************************************************
*                     SEGGER Microcontroller GmbH                    *
*                        The Embedded Experts                        *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
File    : OS_StartLEDBlink.c
Purpose : embOS sample program running two simple tasks, each toggling
          a LED of the target hardware (as configured in BSP.c).
*/

#include "RTOS.h"
#include "BSP.h"

static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
static OS_TASK         TCBHP, TCBLP;                // Task control blocks

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_TASK_Delay(200);
  }
}

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  BSP_Init();   // Initialize LED ports
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();   // Start embOS
  return 0;
}

/************************** End of file ****************************/
```


1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screen shot below). If the debugger does not halt at the `main()` function, set a breakpoint at the first instruction in the `main()` function.



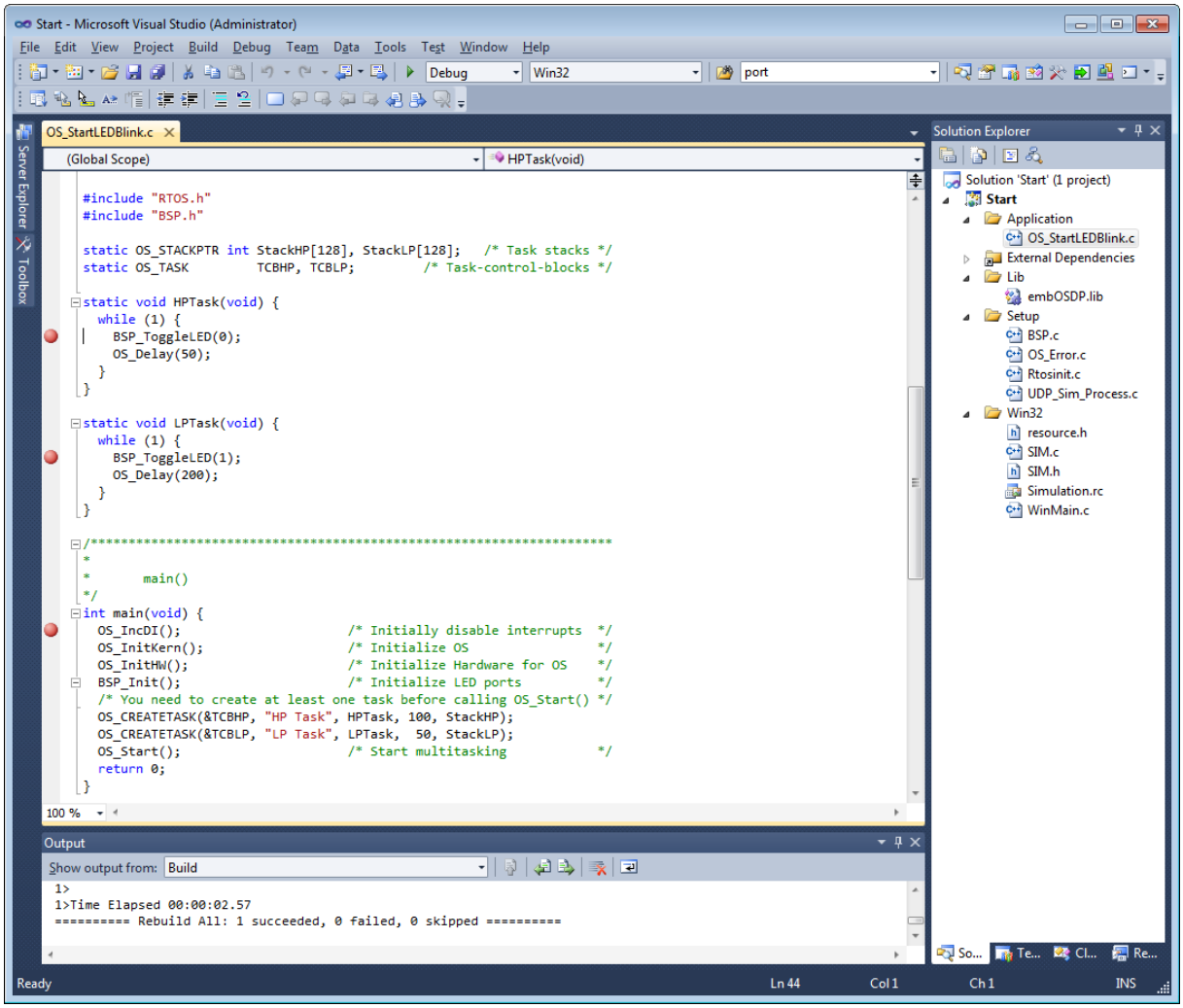
The embOS simulation environment is set up and displays a simulated device which is shown before the debugger stops at the breakpoint at `main()`.



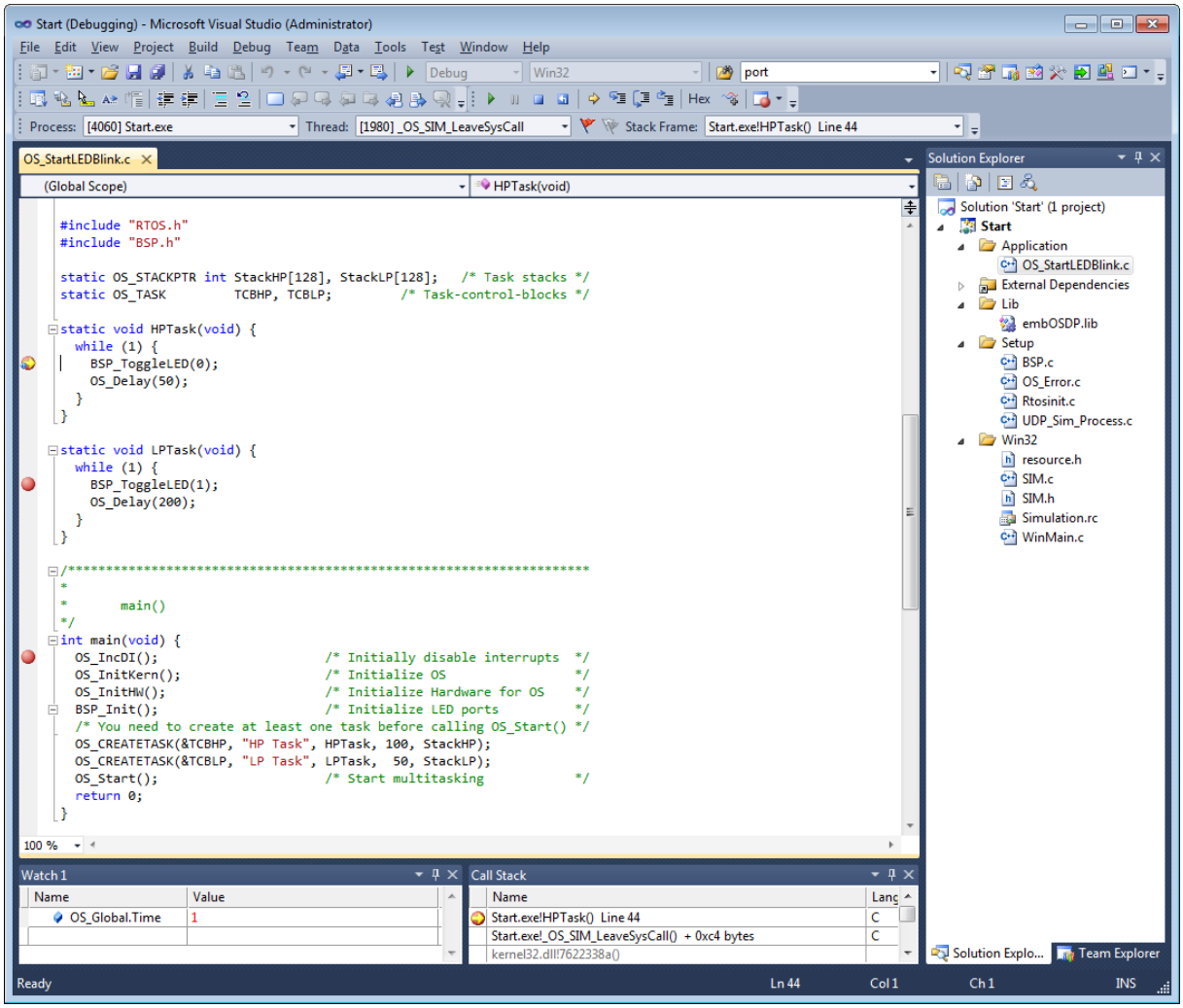
`OS_Init()` is part of the embOS library; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`OS_InitHW()` is part of `RTOSInit.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

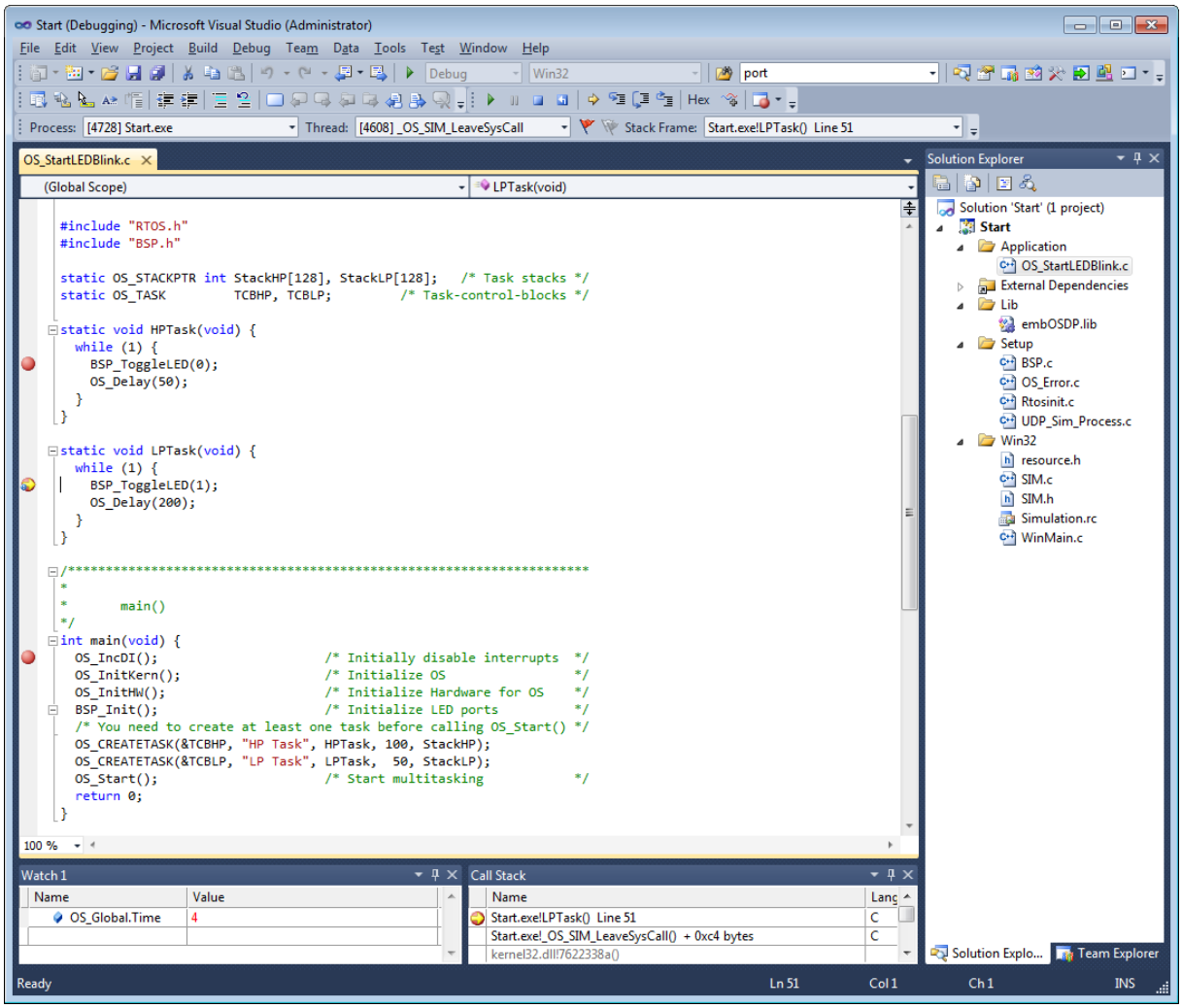
`OS_Start()` should be the last line in `main()`, because it starts multitasking and does not return, unless `OS_Stop()` is used. Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.



Step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task. If you continue stepping, the first LED of your device will be switched on, the `HPTask()` will run into its delay and therefore, embOS will start the task with lower priority.



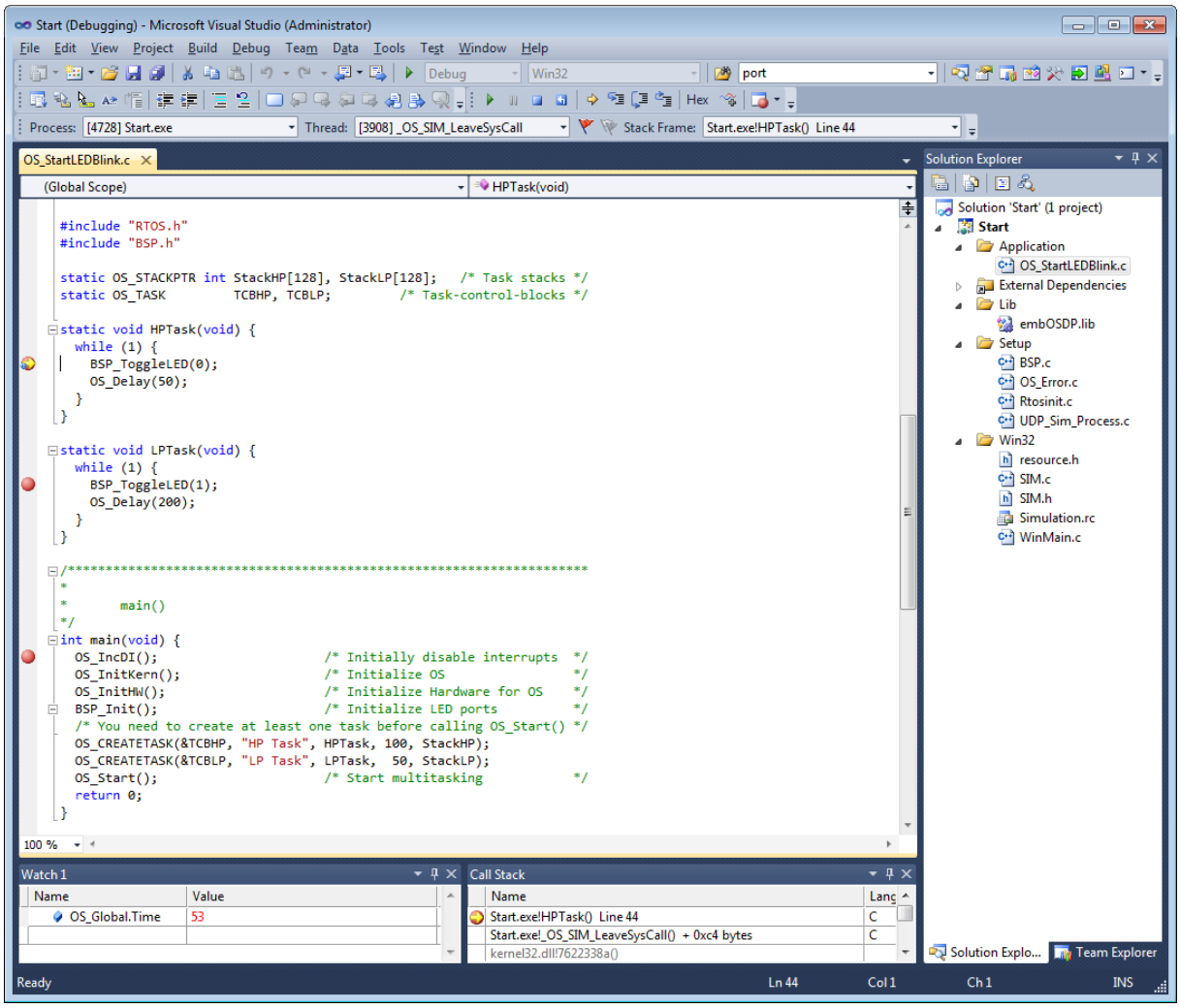
Continuing to step through the program, the `LPTask()` will switch on LED1 and then run into its delay.



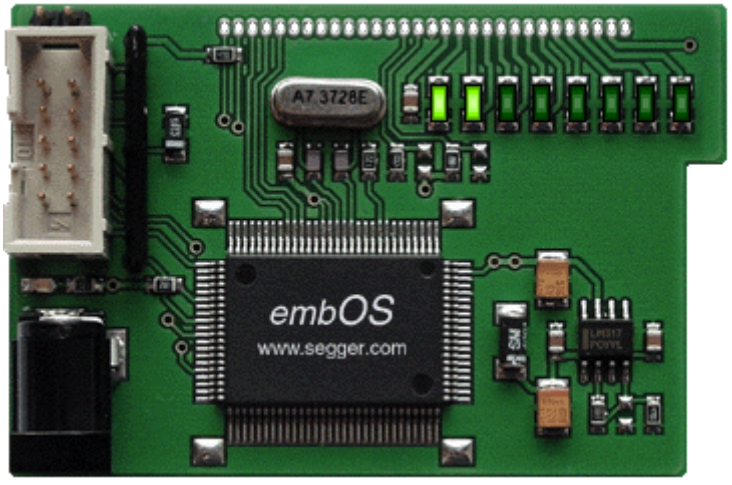
As there is no other task ready for execution when `LPTask()` runs into its delay, embOS will suspend `LPTask()` and switch to the idle process, which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

The embOS Simulation does not contain an `OS_Idle()` function which is implemented in normal embOS ports. Idle times are spent in the Windows idle process i.e. all threads are suspended or sleeping.

When you step over the `OS_TASK_Delay()` function of the `LPTask()`, you will arrive back in the `HPTask()`. As can be seen by the value of embOS timer variable `OS_Global.Time`, shown in the Watch window, `HPTask()` continues operation after expiration of the 50 system tick delay.



Please note, that delays seem to be longer than expected. When the debugger stops at a breakpoint, it takes some time until the screen is updated and the `OS_Global.Time` variable is examined. Therefore `OS_Global.Time` may show larger values than expected. After the delay of `HPTask()` expired, the simulated device shows both LEDs lit as can be seen in the following screenshot:



You may now disable the two breakpoints in our tasks and continue the execution of the application to see how the simulated device runs in real time.

Chapter 2

Build your own application

2.1 Introduction

This chapter provides all information to set up your own embOS simulation project. To build your own application, you should always start with the supplied sample project. Therefore, select the embOS sample project as described in chapter *First Steps* on page 11 and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

To build an application using embOS simulation, the following files from your embOS distribution are required and have to be included in your project:

File	Usage
`Start\Lib\`	
`embOS*.lib`	One of the embOS libraries.
`Start\Inc\`	
`RTOS.h`	Declares all embOS API functions and data types and has to be included in any source file using embOS functions.
`Start\BoardSupport\Simulation\Setup\`	
`RTOSInit.c`	Contains initialization code for the embOS timer interrupt handling and simulation.
`OS_Error.c`	Contains the `OS_Error()` function that is called when an application error occurs.

In addition, if the hardware visualization provided by SEGGER is desired, following files need to be added to your project:

File	Usage
`Start\BoardSupport\Simulation\Win32\`	
`WinMain.c`	Calls the embOS simulation window initialization function and then jumps into the main() entry function.
`SIM_OS.c`	Contains initialization and update functionality of the simulated device window. This file may be modified to support your own simulated device as described later in this manual.
`SIM_OS.h`	Contains declarations for initialization and update functionality of the simulated device window.
`SIM_OS_Device.bmp`	Image of a hardware device used for the Win32 window.
`SIM_OS_Resource.h`	Contains resource defines used for the Win32 window.
`SIM_OS_Simulation.ico`	Application icon.
`SIM_OS_Simulation.rc`	Resource file used for the Win32 window.

All files, API functions and definitions related to the pure visualization of the simulation have the "`SIM_OS_*`" name scheme and are not needed by the embOS simulation to run properly.

2.3 Select a start project configuration

The embOS simulation comes with start project which includes four configurations:

Configuration	Description
Start - Win32 Debug	32-Bit configuration used for development and debugging.
Start - Win32 Release	32-Bit configuration used to build a release executable. It may be used for demonstration purposes.
Start - x64 Debug	64-Bit configuration used for development and debugging.
Start - x64 Release	64-Bit configuration used to build a release executable. It may be used for demonstration purposes.

2.4 Add your own code

For your own code, you may add a new folder to the project or add additional files to the `Application` folder. You may modify or replace the sample application source file in the `Application` directory.

The `main()` function has to be used as an entry point of the embOS simulation. Your `main()` function has to initialize embOS by calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.5 Rebuilding the embOS libraries

New libraries for the embOS simulation can only be built using the source version of the embOS simulation.

- Modify the `Prep.bat` batch file in the root directory of the embOS Simulation source distribution to set the path to the vcvars32.bat and vcvars64.bat developer command files.
- Finally start `M.bat` to produce a new `Start\Lib\` folder which then contains the new libraries.

Chapter 3

Real-time Behavior

3.1 Real-time Behavior

In contrast to bare-metal applications on embedded devices, the embOS simulation is only one of many processes that are executed concurrently on Windows and the underlying CPU. Therefore, it is worth emphasizing that the embOS simulation relies on Windows and its scheduler. The embOS simulation can only execute if Windows tells it to. When working on a PC with not enough performance and/or many other processes running concurrently, it is likely that another process gets scheduled instead of the embOS simulation at some point. This can result in different behavior of the simulated application.

Chapter 4

Device Simulation

This chapter describes how the device simulation works and how it is used.

4.1 Introduction

The embOS simulation contains additional functions to visualize and simulate a hardware device. This may be used to simulate and visualize hardware ports, LEDs or displays which would normally be controlled with functions running on your real hardware. The following chapter describes how the device simulation works in our sample project and how this simulation can be modified to simulate your own hardware.

4.2 How the device simulation works

During startup of the application in `WinMain()`, `SIM_OS_InitWindow()` is called, which creates an event as well as a Windows thread and then waits for the Event to be signaled. The created thread initializes the device window and handles its window messages. As soon as the windows is displayed on the screen, the event will be signaled and the main thread continues its execution. At last, the main thread jumps into the main() function, which represents the main entry point of the simulated application.

In order to simulate the embOS on a single core device, the Windows Affinity Mask is set by `OS_Init()`, so that the process runs on a single core of the PC only. Furthermore, it deactivates HyperThreading for the process, to prevent unexpected program execution. After `OS_Start()` is called, the main thread will be used by the embOS scheduler. The scheduler, again, creates a Windows thread for each created embOS task and handles their execution.

4.3 Immediate device update

The function SIM_OS_UpdateWindow() from SIM_OS.c may be called to force an immediate update of the simulated device. This function does not take parameters and does not return any value.

For an example, take a look at our BSP.c file, which is responsible for toggling the LEDs. Our sample device consists of 8 LEDs which can be switched on, off or can be toggled by the sample hardware specific functions defined in this sample code file. After calling any of those functions, the SIM_OS_UpdateWindow() routine is called to redraw the device window as soon as possible.

```
/*********************************************************************
*
*       BSP_SetLED()
*/
void BSP_SetLED(int Index) {
  if (Index < 32) {
    LEDs |= (1u << Index);
    SIM_OS_UpdateWindow();
  }
}

/*********************************************************************
*
*       BSP_ClrLED()
*/
void BSP_ClrLED(int Index) {
  if (Index < 32) {
    LEDs &= ~(1u << Index);
    SIM_OS_UpdateWindow();
  }
}

/*********************************************************************
*
*       BSP_ToggleLED()
*/
void BSP_ToggleLED(int Index) {
  if (Index < 32) {
    LEDs ^= (1u << Index);
    SIM_OS_UpdateWindow();
  }
}
```


4.4 Periodical device update

Too many calls of SIM_OS_UpdateWindow() may increase the CPU load of the application. In those cases redrawing the device window periodically might be more appropriate. To do so, just modify the SIM_OS_TIMER_PERIOD macro at the top of the SIM_OS.c file. The value defined by the macro represents the timer period in milliseconds. If the period is zero, which is the default value, no timer will be created an used.

4.5 How to use your own simulated device

Any Windows bitmap file can be used and shown as main window of the embOS simulation to visualize a simulated device. To display the real contour of the device, the background outside the device contour may be filled with one specific color, which is treated as transparent when the bitmap is shown on the screen. This “transparent” color should of course not exist in the device itself.

To display your device on the screen during simulation, proceed as follows:

- Create a bitmap file of your device.
- Follow the contour of your device and fill the background with one color, that does not exist in your device image to make the background transparent. Per default, we use pure red in our sample.
- Save the device as SIM_OS_Device.bmp in the directory BoardSupport\Simulation\Win32 of your start project.
- In SIM_OS.c check or modify the define OS_SIM_TRANSPARENT_COLOR to the desired RGB value which shall be transparent.
- To simulate visual elements of your device, write or modify the SIM_OS_PaintWindow() function in SIM_OS.c, as well as _WndProc() callback function to handle special events.

Finally, you have to write your own “hardware” file similar to our BSP.c sample file which transforms your hardware outputs to any memory variables or states which can be visualized by your SIM_OS_PaintWindow() function.

Chapter 5

Libraries

5.1 Naming conventions for prebuilt libraries

embOS is shipped with different pre-built libraries with different combinations of features. The libraries are named as follows:

`embOS<Architecture><LibMode>.lib`

Parameter	Meaning	Values
`Architecture`	Specifies the architecture	: 32-Bit embOS (x86) 64 : 64-Bit embOS (x86_64)
`LibMode`	Specifies the library mode	XR : Extreme Release R : Release S : Stack check SP : Stack check + profiling D : Debug DP : Debug + profiling + Stack check DT : Debug + profiling + Stack check + trace

Example

`embOSDP.lib` is the library for 32-Bit embOS simulation with debug and profiling support.

`embOS64DP.lib` is the library for 64-Bit embOS simulation with debug and profiling support.

Chapter 6

Stacks

The following chapter describes stack handling of the embOS simulation running under Windows.

6.1 Task stacks

Every embOS task has to have its own stack. Task stacks can be located in any RAM memory location. In embOS Simulation, every task runs as a separate Windows thread. The real "task" stack is therefore managed by Windows. Declaration and size of task stacks in your application are necessary for generic embOS functions, but do not affect the stack size of the generated Windows thread. A stack check and stack overflows can therefore not be simulated.

6.2 System stack

The system stack used during startup and embOS internal functions is controlled by Windows, because simulated startup and system calls run in a Windows thread. Stack checking of system stack can therefore not be simulated.

6.3 Interrupt stack

Simulated interrupts in the embOS simulation run as Windows thread with higher priority. As the thread's stacks are managed by Windows, the interrupt stacks will never overflow and stack check can not be simulated.

Chapter 7

Interrupts

This chapter describes how interrupts are handled by the embOS simulation.

7.1 Introduction

With the embOS simulation, interrupts have to be simulated and thus differ from those used in your embedded application. The following chapter describes interrupt simulation and handling in the embOS simulation running on Windows.

7.2 How interrupt simulation works

With the embOS simulation, all interrupt handler functions are started as individual Windows threads running at highest priority that is `THREAD_PRIORITY_TIME_CRITICAL`. Because embOS might have to disable interrupts when embOS internal operations are performed, the embOS simulation has to be able to suspend and resume interrupt handler threads. This requires, that all interrupt handler threads have to be created and installed by the special embOS simulation API function `OS_SIM_CreateISRThread()`.

Interrupt simulation under embOS simulation works as follows:

- An interrupt handler is written as a normal "C"-function without parameters or return value. The interrupt handler function should contain an endless loop which calls `SleepEx(x, TRUE)` at the end of the loop, where 'x' is either a delay in milliseconds or `INFINITE` if the interrupt shall only be resumed via the APC queue technique. The code within the loop up to the `SleepEx(x, TRUE)` represents the actual interrupt code.
- The interrupt handler function is initialized and started as a Windows thread running at highest priority by using the embOS API function `OS_SIM_CreateISRThread()`.
- As soon as the interrupt handler calls `SleepEx(x, TRUE)`, the thread is suspended by the Windows operating system.
- The interrupt is either triggered by an "APC" function call which is used to resume the interrupt simulation thread suspended by `SleepEx(x, TRUE)` or waits until the sleep period expired.
- The interrupt handler continues execution, iterates its task within the loop, calls `SleepEx(x, TRUE)` and suspends until it is resumed again by the associated "APC" function call or the sleep period expired.

7.3 Defining interrupt handlers for simulation

Interrupt handlers used in the embOS simulation can not handle the real hardware normally used in your target application. The interrupt handler functions of your real target application have to be replaced by a modified version which can be used in the simulation.

Simple ISR example:

```
static void _ISRTimerThread(void) {
  while (1) {
    OS_INT_Enter();     // Tell embOS that interrupt code is running
    DoTimerHandling();  // Any functionality can be added here
    OS_INT_Leave();     // Tell embOS that interrupt code ends
    SleepEx(10, TRUE);  // Suspend until delay expired or triggered
                        // by "APC" function
  }
}
```

How to create and use a timer object which periodical calls an "APC" function to signal the interrupt thread can be seen in `RTOSInit.c`. There we use a timer object to signal the embOS timer interrupt thread.

> **Note**
>
> Do not forget to call `OS_INT_Enter()` or `OS_INT_EnterNestable()` before any other embOS function is called from the interrupt handler.

7.4 Interrupt priorities

With embOS Simulation, all simulated interrupts, installed with function `OS_SIM_CreateISRThread()` run on the same ISR thread priority, which is `THREAD_PRIORITY_TIME_CRITICAL`. The order of thread activation is scheduled by Windows and can not be influenced or predicted.

7.5 API functions

Routine	Description	main	Task	ISR	Timer
`OS_SIM_CreateISRThread()`	Installs an ISR handler.	●	●	●	●
`OS_SIM_CreateISRThreadEx()`	Installs an ISR handler and sets a name.	●	●	●	●
`OS_SIM_DeleteISRThread()`	Uninstalls an ISR handler.	●	●		

7.5.1 OS_SIM_CreateISRThread()

Description

`OS_SIM_CreateISRThread()` installs an embOS Simulation ISR handler.

Prototype

```
void* OS_SIM_CreateISRThread(OS_ISR_HANDLER* pfStartAddress);
```

```
typedef void OS_ISR_HANDLER(void);
```

Parameters

Parameter	Description
`pStartAddress`	Pointer to void function which serves as simulated interrupt handler.

Return Value

A handle to the created interrupt simulation thread.

Additional Information

The returned thread handle may be used to create and assign a synchronization object for the created thread. An example on how to use this handle for creation of a timer object to periodical signal an interrupt can be seen in our `RTOSInit.c` where it is used for embOS timer interrupt simulation.

> **Note**
>
> In previous versions of embOS, `OS_SIM_CreateISRThread()` returned a different type. Since embOS V5.10.1, instead of an `OS_U32`, `OS_SIM_CreateISRThread()` returns a `void*`.

7.5.2 OS_SIM_CreateISRThreadEx()

Description

`OS_SIM_CreateISRThreadEx()` installs an embOS Simulation ISR handler.

Prototype

```
void* OS_SIM_CreateISRThreadEx(      OS_ISR_HANDLER* pfStartAddress,
                               const char*           sThreadName);
```

```
typedef void OS_ISR_HANDLER(void);
```

Parameters

Parameter	Description
`pStartAddress`	Pointer to void function which serves as simulated interrupt handler.
`sThreadName`	Interrupt handler name.

Return Value

A handle to the created interrupt simulation thread.

Additional Information

The returned thread handle may be used to create and assign a synchronization object for the created thread. An example on how to use this handle for creation of a timer object to periodical signal an interrupt can be seen in our `RTOSInit.c` where it is used for embOS timer interrupt simulation.

> **Note**
>
> In previous versions of embOS, `OS_SIM_CreateISRThreadEx()` returned a different type.
> Since embOS V5.10.1, instead of an `OS_U32`, `OS_SIM_CreateISRThreadEx()` returns a `void*`.

7.5.3 OS_SIM_DeleteISRThread()

Description

`OS_SIM_DeleteISRThread()` uninstalls an embOS Simulation ISR handler.

Prototype

```
void OS_SIM_DeleteISRThread(void* pThreadHandle);
```

Parameters

Parameter	Description
`ThreadHandle`	Handle to the ISR which was returned by `OS_SIM_CreateISRThread()`.

Additional Information

This function will block the current thread, until the ISR thread has been terminated. In order to uninstall an ISR, it has to be ensured that the ISR thread is scheduled by windows after calling `OS_SIM_DeleteISRThread()`, so that the thread will be terminated properly.

> **Note**
>
> In previous versions of embOS, `OS_SIM_DeleteISRThread()` expected the parameter `pThreadHandle` to be of a different type.
> Since embOS V5.10.1, instead of an `OS_U32`, `OS_SIM_DeleteISRThread()` expects a `void*`.

Example

The following example shows an ISR tick handler and how it can be uninstalled by adding a new function `DeInitSystemTick()`. `DeInitSystemTick()` can then be called after `OS_Stop()` and `OS_DeInit()`.

```
static HANDLE _hISRThread;
static UINT   _hTimerEvent;

/*********************************************************************
*
*       _ISRTickThread()
*/
static void _ISRTickThread(void) {
  //
  // ... initialization
  //
  while (1) {
    //
    // ... system tick is handled here
    //
    SleepEx(INFINITE, TRUE);  // Blocks until an APC is queued
  }
}
```



```
/*********************************************************************
*
*       _voidAPC()
*
*  Function description
*    Dummy APC function. Is required because we (ab)use the
*    WIN32 QueueUserAPC() API function to wake up a thread
*/
static void APIENTRY _voidAPC(DWORD Dummy) {
  OS_USEPARA(Dummy);
}

/*********************************************************************
*
*       _CbSignalTickProc()
*
*  Function description
*    Timer callback function which periodically queues an APC in order
*    to resume the ISR tick thread.
*/
static void CALLBACK _CbSignalTickProc(UINT uID, UINT uMsg, DWORD dwUser,
                                       DWORD dw1, DWORD dw2) {
  OS_USEPARA(uID);
  OS_USEPARA(uMsg);
  OS_USEPARA(dw1);
  OS_USEPARA(dw2);
  QueueUserAPC(_voidAPC, (void*)dwUser, 0);
}

/*********************************************************************
*
*       OS_InitHW()
*
*  Function description
*    Initialize the hardware required for embOS to run.
*/
void OS_InitHW(void) {
  //
  // ...
  //
  _hISRThread  = (HANDLE)OS_SIM_CreateISRThread(_ISRTickThread);
  _hTimerEvent = timeSetEvent(1, 0, _CbSignalTickProc, (int)_hISRThread,
                              (TIME_PERIODIC | TIME_CALLBACK_FUNCTION));
  //
  // ...
  //
}

/*********************************************************************
*
*       DeInitSystemTick()
*
*  Function description
*    De-initialize the system tick handler.
*/
void DeInitSystemTick(void) {
  //
  // Delete the ISR handler.
  //
  OS_SIM_DeleteISRThread(_ISRTickThread);
  //
  // Delete the timer which activated the ISR thread periodically
  // by queuing APCs.
  //
  timeKillEvent(_hTimerEvent);
}
```


Chapter 8

Handling Windows Callbacks

8.1 Handling Windows Callbacks

Windows callback functions may sometimes need to use embOS API, for instance in order to signal an embOS task. Unfortunately, embOS API must not be used directly from such callback functions. However, it is possible to trigger a simulated interrupt from within the callback function and subsequently call the desired embOS API from within that interrupt. For example, this could be done by using the APC mechanism described within the chapter *Interrupts* on page 31.

Example

```
static HANDLE _hISRThread = NULL;

/*********************************************************************
*
*       _Interrupt()
*/
static void _Interrupt(void) {
  while (1) {
    SleepEx(INFINITE, TRUE);
    OS_INT_Enter();
    ...
    OS_INT_Leave();
  }
}

/*********************************************************************
*
*       _voidAPC()
*
*  Function description
*    Dummy APC function. Is required because we (ab)use the
*    WIN32 QueueUserAPC() API function to wake up a thread
*/
static void APIENTRY _voidAPC(ULONG_PTR pDummy) {
  OS_USEPARA(pDummy);
}

/*********************************************************************
*
*       SomeWindowsCallback()
*
*  Function description
*    This function gets called directly by Windows.
*/
static void CALLBACK SomeWindowsCallback(void) {
  if ((_hISRThread != NULL) && (OS_IsRunning() != 0)) {
    QueueUserAPC(_voidAPC, _hISRThread, 0);
  }
}

int main(int argc, char* argv[]) {
  ...
  //
  // Create the interrupt
  //
  _hISRThread = (HANDLE)OS_SIM_CreateISRThread(_Interrupt);
  OS_Start();
}
```


Chapter 9

Calling blocking non-embOS functions from tasks

This chapter describes how to use non-embOS functions which might be blocking in the embOS simulation.

9.1 Introduction

With the embOS simulation, typically the embedded application shall be simulated. Calling non-embOS functions from tasks, however, may be required for several reasons, e.g. with a simulated TCP/IP stack that uses the Windows socket interface for communications via the PC. Calling blocking embOS functions will suspend the task for the time it is waiting and allows tasks with lower priority to be scheduled by embOS. Calling blocking non-embOS functions will freeze the calling task and no other task with lower priority will be scheduled. This may cause the whole simulation to stop until the blocking task continues execution. To avoid this, two embOS API functions are available to manage the call of blocking non-embOS functions.

Similar to handling critical regions, there is one entry function (`OS_SIM_EnterSysCall()`, which has to be called before calling a blocking non-embOS function) and one exit function (`OS_SIM_LeaveSysCall()`, which has to be called after execution of the blocking non-embOS function).

The Application folder of the embOS shipment contains the sample application `OS_SimBlocked.c`, which demonstrates these functions' usage on blocking Windows API calls.

9.2 API functions

Routine	Description	main	Task	ISR	Timer
`OS_SIM_EnterSysCall()`	Must be called prior to calling any blocking non-embOS function from a task.		•		
`OS_SIM_LeaveSysCall()`	Must be called after calling any blocking non-embOS API function from a task, and before any other embOS API function is called.		•		

9.2.1 OS_SIM_EnterSysCall()

Description

`OS_SIM_EnterSysCall()` has to be called before a blocking non-embOS function is called from a task.

Prototype

```
void OS_SIM_EnterSysCall(void);
```

Additional information

`OS_SIM_EnterSysCall()` has to be called before calling any blocking non-embOS function.

After calling `OS_SIM_EnterSysCall()`, no further embOS API function except `OS_SIM_EnterSysCall()` or `OS_SIM_LeaveSysCall()` must be called.

Nested calls of `OS_SIM_EnterSysCall()` are allowed. The function must only be called from tasks.

Example

```
...
OS_SIM_EnterSysCall();
// Any blocking non-embOS function may be called now...
...
recv (socket, pBuf, len, flags);
// Any other code may follow.
// No embOS function must be called except OS_SIM_LeaveSysCall()
...
OS_SIM_LeaveSysCall();
// From now on, calling other embOS functions is allowed
...
```


9.2.2 OS_SIM_LeaveSysCall()

Description

OS_SIM_LeaveSysCall() has to be called after execution of a blocking non-embOS function, before any other embOS function is called.

Prototype

```
void OS_SIM_LeaveSysCall(void);
```

Additional information

OS_SIM_LeaveSysCall() has to be called after execution of blocking non-embOS API functions, and before calling any other embOS function.

It must be called only when OS_SIM_LeaveSysCall() has been called by the same task before. For every call of OS_SIM_EnterSysCall(), OS_SIM_LeaveSysCall() has to be called a corresponding number of times. After the last call to OS_SIM_LeaveSysCall(), the task is switched back to normal embOS execution mode.

Nested calls of OS_SIM_LeaveSysCall() are allowed. The function must only be called from tasks.

Example

```
...
OS_SIM_EnterSysCall();
// Any blocking non-embOS function may be called now...
...
recv (socket, pBuf, len, flags);
// Any other code may follow.
// No embOS function must be called except OS_SIM_LeaveSysCall()
...
OS_SIM_LeaveSysCall();
// From now on, calling other embOS functions is allowed
...
```


Chapter 10

Updating the Simulation

This chapter provides all information on how to update the embOS simulation with regards to the Win32 BSP files, which are used for the graphical visualization. For information on how to update embOS to a newer version, please refer to the chapter “Update” of the generic embOS manual.

10.1 Updating the Win32 BSP files

The Win32 BSP files allow for a graphical visualization of the simulation, to indicate whether the simulation is currently executing. By default, a window containing the outline of a circuit board will be created. This board contains some LEDs that can be turned on and off using the BSP functions contained within the source file `BSP.c`. These files are optional to the embOS simulation and may therefore be modified and extended in functionality, or even removed if desired. Therefore, updating those files typically is not necessary. If updating the Win32 BSP files is desired, however, two cases may apply:

The old Win32 BSP files were not modified

In this case, any previous Win32 files may simply be removed and replaced by all files contained in the Win32 directory of the updated shipment.

The old Win32 BSP files were modified

In this case, any previous Win32 files should be merged with their respective counterparts from the updated shipment in order to include the performed customizations with the new files as well.

Chapter 11

Technical data

11.1 Resource Usage

The memory requirements of embOS for RAM differs depending on the used features, CPU, compiler, and library model. The following values are measured using embOS library mode `OS_LIBMODE_XR`.

Module	Memory type	Memory requirements
embOS kernel	RAM	~136 bytes
Task control block	RAM	60 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	16 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLock	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes

